Hit List

Clear | Generate Collection | Print | Fwd Refs | Bkwd Refs | Generate OACS

Search Results - Record(s) 1 through 1 of 1 returned.

☐ 1. Document ID: US 6430685 B1

L1: Entry 1 of 1 File: USPT Aug 6, 2002

US-PAT-NO: 6430685
DOCUMENT-IDENTIFIER: US 6430685 B1

TITLE: Method and apparatus for enabling a computer system

DATE-ISSUED: August 6, 2002

INVENTOR-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY
Yu; Dean T.	Cupertino	CA		
Derossi; Christopher S.	San Jose	CA		

ASSIGNEE-INFORMATION:

NAME	CITY	STATE	ZIP CODE	COUNTRY	TYPE CODE
Apple Computer, Inc.	Cupertino	CA			02

APPL-NO: 08/ 558929 [PALM]
DATE FILED: November 13, 1995

PARENT-CASE:
RELATED APPLICATIONS This is a continuation of co-pending application Ser. No. 08/019,599 filed on Feb. 19, 1993.

INT-CL: [07] G06 F 9/24, G06 F 9/445

US-CL-ISSUED: 713/1; 713/100
US-CL-CURRENT: 713/1; 713/100

FIELD-OF-SEARCH: 395/700, 395/681, 395/651, 395/652, 395/653, 713/1, 713/2, 713/100, 711/163, 711/170, 709/100, 709/101, 709/102, 709/103, 709/104

PRIOR-ART-DISCLOSED:

U.S. PATENT DOCUMENTS

PAT-NO	ISSUE-DATE	PATENTEE-NAME	US-CL
3618045	November 1971	Campbell	395/650
4493034 ✓	January 1985	Angelle et al.	395/700

h e b b g e e e f e egc c ef b e

4558413	December 1985	Schmidt et al.	364/300
4620273	October 1986	Mitani et al.	364/136
4626986	December 1986	Mori	395/700
4654783	March 1987	Veres et al.	364/200
4833594	May 1989	Familetti et al.	395/700
5128995	July 1992	Arnold et al.	380/4
5134580	July 1992	Bertram et al.	395/650
5214771	May 1993	Clara et al.	395/500
5261104	November 1993	Bertrhm et al.	395/700
5278973	January 1994	O'Brien et al.	395/700
5313626	May 1994	Jones et al.	395/575
5325430	June 1994	Smyth et al.	380/4
5459854	October 1995	Sherer et al.	395/500

OTHER PUBLICATIONS

Rose, Philip F.H., The Macintosh Finder: Pure GUI, PC Magazine Sep. 12, 1989 V8 n15 p 133(4).*
Wiggins, Robert R., Vnemotional Instability, Mac User 6/90 V6 n6 p 29(1).

ART-UNIT: 2151

PRIMARY-EXAMINER: Banankhah; Majid A.

ATTY-AGENT-FIRM: Blakely, Sokoloff, Taylor & Zafman

ABSTRACT:

A system and method for utilizing generic computer operating system software for computer hardware systems designed subsequent to the operating system software. The system and method of the present invention employ a separate modular software file called a System Enabler that has all patches, code, data and resources needed to make a particular computer system operational. The System Enabler file is matched to a particular hardware system and may be bundled with that hardware system. During computer system start up the System Enabler file modifies the generic operating system software for optimum operation with the particular computer hardware system.

19 Claims, 5 Drawing figures

Term	Documents
"6430685"	1
6430685S	0
"6430685".PN..USPT.	1
(6430685.PN.).USPT.	1

h e b b g e e e f e egc c ef b e

The

L29: Entry 1 of 1 File: USPT Jan 8, 1985

DOCUMENT-IDENTIFIER: US 4493034 A

** See image for Certificate of Correction **

TITLE: Apparatus and method for an operating system supervisor in a data processing system

Brief Summary Text (5):

It is known in the related art to provide a data processing system capable of execution of an instruction set under control of a single operating system. Each operating system has an interior decor, including a distinctive address formation and instruction processing characterists, that prohibit easy portability of operating systems. In order to utilize a plurality of operating systems, it has been necessary to alter the operating systems or add additional apparatus to operate additional apparatus. In addition, it is frequently necessary to reinitialize the data processing system each time a different operating system was activated. Frequently, improvements in operating systems require a change in apparatus and can cause problems in data systems. In addition, fault procedures residing in the memory unit have the characteristics of a specialized operating system.

Brief Summary Text (12):

It is yet another object of the present invention to provide a procedure for terminating operation of a currently active operating system and initialize operation of a different operating system.

Detailed Description Text (9):

Referring to FIGS. 2 and 3, the steps in inactivating a currently active operating system (1) and activating another operating system is illustrated. In step 201, the data processing system is currently executing instructions under the control of operating system 1. A fault or interrupt condition for supervisor procedures is identified in step 202. The identified condition must be a preselected condition in which the result is an exchange of operating systems controlling the data processing system. The preselected fault or interrupt condition will cause a predetermined location in the reserved memory space of operating system 1 to be addressed indicated as step 203 in FIG. 2. This process is shown as path 301 from operating system 1 memory 312 to operating system reserve memory 301. The location in reserve memory has a series of steps, the most significant being the storage of register contents in the central processing unit, step 205, the register parameters will be stored in reserve memory so that upon restoration of operating system 1 as the currently active operating system, the data processing unit will be returned to this state. After execution of the fault entry programs, a location in switching entry section of the operating system reserve memory 1 is executed. The instructions in this location allow the operating system 2 reserve memory to be addressed in a location in the operating system 2 switching entry location, i.e., step 206. This transfer to operating system 2 is shown as path 302 in FIG. 3. The instructions in the switching entry portion of operating system 2 cause, in step 207, the stored parameters of operating system 2 to be entered in the appropriate registers in the data processing thus initializing the system or restoring the system to the last previous state of operating system 2. The final step 208 and

h e b b g e e e f c e egc c e ge

path 303 transfers control of the central processing unit to operating system 2. In this manner, the control of the central processing unit 1 has been transferred from operating system 1 to operating system 2. In FIG. 3, paths 304, 305, and 206 illustrate the process by which control of the central processing unit 1 is transferred back to operating system 1. Also shown in FIG. 3 is the possibility that each operating system can control a plurality of central processing units. For that situation, the other central processing units will switch between operating systems in a similar manner.

Detailed Description Text (10):
In dealing with a data processing unit sharing operating systems, it is necessary that memory space alloted to each operating system be inaccessible to the other operating systems. FIG. 4 indicates the manner in which this may be accomplished. A page table for the operating system is shown with locations associated with each of two operating systems. The address associated with operating system 1 will point to a group of locations in the physical memory 403. Similarly, the addresses stored in the group of locations associated with operating system 2 indicate a different group of memory locations. Thus, the physical memory 403 is divided in groups of memory addresses which are the result of a paging operation in the page table 402. Thus, operating system 1, or operating system 2, can be addressed in contiguous locations (i.e., for each operating system), but the groups of physical memory addresses associated with each page table address can be located throughout the physical memory address space. One advantage of the paging is that "holes" in the physical memory space (i.e., such as can result from an error in the memory unit) can be avoided when the page table is formed during an initialization process. FIG. 4 also illustrates the important feature that the reserved memory spaces 410 and 411 for operating system 1 and operating system 2 are unavailable to the operating system. Furthermore, the reserve memory spaces can be located anywhere in the physical memory.

Detailed Description Text (11):
Referring next to FIG. 5, the use of the supervisor base address and the supervisor bound for the isolation of operating system is shown. A supervisor page table directory 764 is provided during initialization, and provides the correspondence between an operating system address and an address in physical memory. During initialization or re-activation of each operating system, the supervisor base register 761 and the supervisor bound register 762 have data entered therein. In this preferred embodiment the first twelve bits of an address developed during normal execution of instruction provides an off-set from the base address. The base address is determined by the currently active operating system, i.e., each operating system will have a predetermined base address in the supervisor page table directory. The supervisor bound quantity will determine the number of page table directory location allocated to the operating system. Thus, when a 26 bit real address 763 is applied to the supervisor paging apparatus, the first 12 bits point to location 770, i.e., the address in the page address register plus the off-set defined by the highest order bit of the real address. The quantity in the bound register ensures that the location 770 is with contiguous directory location allocated to the operating system. The contents of location 770 are a 12 bit quantity which replaces the 12 bit off-set quantity in the address to provide an address in physical memory.

Detailed Description Text (12):
Referring to FIG. 6, the general format of the reserved memory 650 for each operating system is shown. The supervisor switcher 651 portion of the reserved memory contains the program necessary for interruption or for initialization of an operating system. A more detailed description of this memory area will be given. Reserved memory portion labelled interrupt queues 652 are hardware loaded prioritized interrupts that are received by an operating system which is currently inactive. When the operating system becomes active, these queues will be interrogated and appropriate responses enabled. Hardware configuration 653 portion

of the reserved memory is loaded upon initialization and provides a record of the resources (i.e., data processing system components) available to the operating system. The connect tables 654 portion of reserved memory provides a list of the resources currently available to the operating systems. Summarizing, the reserved memory is devoted to storing information necessary to maintain the isolation of the operating systems.

Detailed Description Text (13):
Referring to the supervisor switches 651 portion of reserved memory 650, the data stored therein provides the coded signals to process the change from one operating system to a second operating system. Included therein are the entry location (into the reserved memory switches) and the exit location from the reserved memory. Locations are included to safestore the contents of the central processing unit registers. Thus safestore locations are loaded by the initialization procedures, and when the operating system is activated, these locations provide the initialization. When the operating system asssociated with reserve memory is inactivated, these locations are filled with the contents of the central processing unit so that when the operating system is reactivated, the central processing unit will return to the state existing at the time it was inactivated. Also stored in the supervisor switches of the reserved memory are the quantities for the supervisor base and the supervisor bound. During initialization of a central processing unit, a supervisor page table direction is established. Each supervisor page generally consists of a multiplicity of normal pages. Moreover, the supervisor page table directory is used in the final translation from the address used by the operating system to the location in physical memory is performed through the supervisor page table. The supervisor base address points to the first in a series of sequential locations in the supervisor page table location. The operating system address contains an offset which indicates which location in the sequential supervisor page table directory the address refers. The supervisor bound ensures that off-set does not exceed the locations in the supervisor page table directory allocated to the operating system. If this occurred, the physical memory location addressed would be outside the area reserved for the operating system. The supervisor base and bound address are stored in registers in the central unit pipeline structure when the operating system is activated. The reserved memory supervisor switches includes a reserved memory base address and a bound. Again, these quantities are stored in the central unit pipeline structure (in the descriptor stack in the preferred embodiment) and provides the address of reserved memory when a predetermined fault is detected. Indeed in the preferred embodiment, the off-set from the base address in the reserved memory for handling of faults requiring attention of the supervisory processes is the same off-set from the operating base addressed as is used in the normal fault handling procedures. Also included in the reserved memory is the code allowing entry into the connect table. A memory location holds a quantity that will be loaded in a supervisor fault enable register. The quantity has a pattern which defines all the fault conditions that require a response from the supervisor procedures. When a fault condition is defined by signal pattern, this pattern is compared with the contents of the supervisor fault enable register to determine if the supervisor procedures or the normal operating procedures should be used to respond to the condition. A reserved memory location is used to store the faults pending register. These contents are re-entered into faults pending register when the operating system is reactivated so that conditions originally existing for the operating system are restored. Other reserved memory locations implement the use of a supervisor timer so that at the end of a predetermined number of clock cycles, the currently active operating system will receive a fault condition causing a new operating system to be activated. Thus one of the locations will have a (clock) count determining the time that the operating system will be active. Still another location contains data to be entered in the option register. This register contains signals control certain decor dependent variables. For example, a decor code is included. This code is compared in an operation code to ensure that the instruction is permitted in the decor of the operating system. Another quantity that can be stored is whether

h e b b g e e e f c e egc c e ge

change The Config

intermediate paging (not the supervisor paging) is employed. Other register locations include other decor dependent data that are loaded into the hardware apparatus to make the central processing unit decor consistent with the active operating system. For example, in virtual address formation, a working space number is required, while other operating systems may not require this quantity to be available to the operating system i.e., loaded in preselected register locations. Finally, an offset for the hardware configuration table is present.

Detailed Description Text (15):
A more detailed use of the supervisor procedures will now be given by way of illustration. When appropriate conditions occur in the central processing unit, such as the supervisor clock reaching a predetermined number of counts, the operating system reaching a point where it will voluntarily relinquish control of the central processing unit etc, a set of signals indicating a fault condition is entered in the fault register. The signals in the fault register are compared with signals that have been previously entered in the supervisor fault enable register. When a coincidence is detected, a different operating system is to be activated through the supervisor procedures, and a supervisor fault procedure is enabled. The supervisor fault procedure uses the reserved memory base address, stored in the central unit pipeline structure (i.e., in the descriptor stack) combined in a constant offset number to enter the reserved memory unit of the currently active operating system. The procedures beginning with the entry address cause the storage quantities, in registers in the central processing unit, in the appropriate locations in the reserved memory. In addition, the contents of the faults pending register are stored. These quantities allow the operating system being deactivated, to resume in the state when the operating system is reactivated. When this storage is complete, the reserved memory exits from a location that addresses an entry in the reserved memory of a second operating system. The second operating system loads the base and bound address of the reserved memory associated with the second operating system into the central unit pipeline structure (i.e., the descriptor stack) so that the instructions executed using the supervisor procedures can have the appropriate address formation. The supervisor base and bound is loaded in the final paging registers thus providing the mechanism for addressing only the physical memory associated with the second operating system and effectively isolating non-associated physical memory from the second operating system. The procedures of the reserved memory load the option register, which in addition to other decor-dependent information, provides the code that determines when a non-permitted instruction (e.g., because of incorrect decor requirements) has entered execution. The procedures of the reserved memory cause the fault pending register to be loaded and the decor-dependent quantities are entered in appropriate registers in the central processing unit. The central processing unit is now either initialized with respect to the second operating system or the previous state, from which the second operating system exited, has been reestablished. The reserved memory of the second operating system now executes a procedure by which the memory associated with the second operating system is entered and control of this data processing unit is now with the second operating system.

First Hit Fwd Refs

End of Result Set

Generate Collection Print

L39: Entry 1 of 1 File: USPT Jul 28, 1992

DOCUMENT-IDENTIFIER: US 5134580 A
TITLE: Computer with capability to automatically initialize in a first operating system of choice and reinitialize in a second operating system without computer shutdown

Brief Summary Text (7):
Once a computer has been initialized and an operating system such as IBM DOS is resident, application programs which run under the operating system can be used. For example, word processors which operate under the operating system can be utilized, spreadsheet programs can be utilized, and information applications such as stock quotations or educational services or User's Club can be run. Should, however the user desire to run an application under a different operating system, it is typically necessary to turn the computer off, insert a different operating system in the diskette drive, turn the computer back on and initialize the computer in the different operating system. Then the application can be inserted in the diskette drive and the application can finally be run.

Brief Summary Text (8):
In order to save time for the user, a personal computer has been developed with the IBM DOS operating system located in read-only-memory (ROM) so that initialization of the machine directly brings the machine up with that operating system ready to use, thus avoiding the need for the computer user to insert a diskette containing an operating system into the diskette drive and/or also avoiding the need for the computer to turn to the fixed disk to find an operating system. However, many users may desire to bring up the machine with a different operating system from that particular version of IBM DOS located in ROM. Therefore, a flexible initialization system has been invented to bypass the operating system located in ROM and thereby enable the user to bring the computer up in an operating system of choice. Additionally, the flexible initialization system enables the user to bring the computer up already in a particular application. Also, provision is made to allow the reading of special DOS start-up files which alter the start-up which would occur utilizing the entirety of the DOS located in ROM. In that manner, flexibility is provided the user so that initialization results in a customized starting point of the user's choice.

Brief Summary Text (9):
It is an object of this invention to provide a computing system which enables the computer user to automatically start the computer in an operating system located in ROM or in the operating system of choice, and then go to a different operating system without turning the computer off and back on.

Brief Summary Text (12):
This invention relates to a computer system in which the user is enabled to automatically start the computer from an operating system stored in ROM or in another operating system of choice. The booting sequence has no dependence on data stored on any diskette or disk and the very first initialization will automatically load the operating system stored in ROM and take the user to a screen with a main menu, "the ROM shell", after booting has completed. However, the system also

includes customizing bits which are stored in permanent read/write memory which can be, for example, battery backed-up complimentary metal oxide semiconductor random access memory (CMOS RAM) in order to allow the user to change the start up sequence. These bits are initially set to a default state, but can be changed by the user. In that manner, the user is enabled a great degree of freedom in initializing the computer to a selected operating system different from the operating system stored in ROM; to a selected application; or to a personalized version of the operating system stored in ROM. This flexible initialization system is altered by this invention to include apparatus for indicating to the system that the computer user desires to end the session and go to still another operating system. Upon recognition of that request, the computer system is reinitialized with the new operating system which will be located in external memory on a flexible diskette or on a fixed disk. Provision is made in the customization word for carrying out the request and for deleting the request thereafter so that the next future initialization is once again in the usual operating system of choice.

Detailed Description Text (5):
As described above, the computing system shown in FIG. 1 contains a version of IBM DOS in ROM which is automatically loaded during the initialization process. That activity speeds the initialization of the system and eliminates the common steps in the prior art of inserting a diskette containing IBM DOS into the diskette drive and then initializing the system. While this new initialization procedure is useful and aids the ease through which the machine is used, at times the user may wish to load a different version of IBM DOS or a completely different operating system such as OS/2, AIX, UNIX or any other operating system suitable for the computing system. As a consequence, it is desirable that the initialization routine be flexible so that these alternate operating systems can be loaded if desired.

First Hit Fwd Refs
End of Result Set

☐ Generate Collection Print

L50: Entry 1 of 1 File: USPT Oct 17, 1995

DOCUMENT-IDENTIFIER: US 5459854 A
TITLE: Method for optimizing software for any one of a plurality of variant architectures

Abstract Text (1):
A method allows a designer to implement software for a wide variety of variant host architectures, without excessive usage of host memory, nor sacrificing the capabilities of high end versions of the variant architectures available. The method is based on providing an initialization module of the software to host memory. A portion of the initialization module determines the host architecture. Based on the determined host architecture, the unneeded portions of the initialization module are freed, and the needed portions are relocated into a contiguous memory space to minimize host memory usage. Any location dependent entries in the needed portions of the program are then updated based on the relocation. The initialization module includes a plurality of code blocks, each of which is optimized to a particular variant architecture. When the variant architecture of the host is identified, those code blocks which are optimized to the identified host are selected and the other code blocks are freed. The selected blocks are then relocated to optimize host memory usage.

Brief Summary Text (5):
The present invention relates to techniques for initializing software, such as device drivers for network interface controllers, in a host data processing system. More particularly, the invention relates to a process for loading a software product in host memory, and then optimizing the product for the particular architecture of the host data processing system.

Brief Summary Text (7):
Computer systems have been implemented according to a large number of variant architectures, where the architecture is defined as the hardware components, such as the CPU type, which make up the system and the software operating system which provides a platform for accomplishing a variety of applications.

Brief Summary Text (8):
The CPU type in a given architecture may include a kind of microprocessor which is used in the CPU. Microprocessor types include the "86" class of microprocessors, originating with and manufactured primarily by Intel Corporation. This class of microprocessors includes the models designated 8086, 8088, 80186, 80286, 80386, 80486, as well as any other microprocessor which shares substantially the same instruction set. Each microprocessor in this family uses a common subset of instructions but have unique capabilities which distinguish them from one another. Other microprocessor classes include the 68000 class provided by Motorola, and a variety of RISC processors made by a number of companies.

Brief Summary Text (9):
Operating system types include most commonly MS-DOS in a variety of versions, developed by Microsoft Corporation of Bellevue, Washington, and variants thereof. Also, the OS/2 operating system, the UNIX operating system and others have been

developed for a wide variety of system architectures.

Brief Summary Text (10):
Thus, an architecture of a data processing system is defined by the hardware which makes up its central processing unit and related components, and the operating system software which provides a platform for running application software and the like. A manufacturer of add-on devices designed to handle a plurality of variant architectures must provide software for controlling the add-on devices which runs in each of the variant architectures. This software is typically referred to as a device driver. Therefore, typically prior art systems required a number of versions of device drivers written for each variant architecture for which the add-on device is intended to run. The user must be able to select which device driver applies to his data processing system, and the manufacturer must distribute a large number of versions. This requires an intelligent user for loading and initializing the device driver.

Brief Summary Text (13):
Computer architectures are typically designed with a limited amount of user accessible random access memory. For instance, the 86 class of microprocessors and the MS-DOS operating system were originally intended to be useful only with a base of 640 kilobytes of user accessible memory. This limitation represents a major memory usage limitation in certain applications of computers employing a given variant architecture.

Brief Summary Text (15):
Therefore, the designer who is intent on distributing software in executable form which must occupy a portion of the user accessible space in a computer system, must account for the variant architectures for which the particular device is intended to be used. One approach of doing so, the operating system is optimized for each particular variant and a selection is made by the user of the correct version of the device driver for his system to be loaded. In another approach, all optional versions are loaded into the limited user accessible space together, thereby using up more space than necessary. Alternatively, the device drivers are written with only the common subset of instructions available to all of the architectures for which the driver is written. In this case, the designer is not able to take advantage of the power and versatility of the more advanced architectures.

Brief Summary Text (16):
Accordingly, there is a need for a process for optimizing software for the particular variant host architecture without sacrificing advanced processes available to the advanced architectures, nor wasting user accessible space with versions of code that are not used in the particular host.

Brief Summary Text (18):
The present invention provides a process which allows a designer to implement software for a wide variety of variant host architectures, without excessive usage of host memory, nor sacrificing the capabilities of high end versions of the variant architectures available.

Brief Summary Text (19):
The method is based on providing an initialization module of the software to host memory. A portion of the initialization module determines the host architecture. Based on the determined host architecture, the unneeded portions of the initialization module are freed, and the needed portions are relocated into a contiguous memory space to minimize host memory usage. Any location dependent entries in the needed portions of the program are updated based on the relocation.

Brief Summary Text (20):
Thus, the initialization module may include a plurality of code blocks, each of which is optimized to a particular variant architecture. When the variant

architecture of the host is identified, those code blocks which are optimized to the identified host are selected and the other code blocks are freed. The selected blocks are then relocated to optimize host memory usage.

Brief Summary Text (21):
According to one aspect of the present invention, the code blocks are organized into a plurality of functional segments, where each segment includes at least one code block optimized for at least one variant architecture. After identification of the host system, a code block from each functional segment in the plurality of functional segments is selected to form the host memory resident portion of the operating system.

Brief Summary Text (22):
In one implementation, the initialization module includes tables for identifying locations of code blocks and of location dependent entries within the code blocks, based on the identified variant architecture. In order to select particular code blocks, the system accesses the tables in response to the identified variant architecture to identify the selected code blocks, accesses the selected code blocks and updates location dependent entries in the code blocks using the tables.

Brief Summary Text (23):
According to another aspect of the invention, the initialization module includes a program for identifying the host architecture. Where the host architecture includes a configuration table, this program identifies the host by parsing the configuration table. Where there is no configuration table, or the configuration memory does not provide enough information, the program executes tests to determine the features of the host architecture for the purpose of identifying the variant in which the program has been loaded.

Brief Summary Text (24):
The present invention is particularly suited to initializing operating system software for controlling an add-on device, such as a network interface driver, where the network interface driver includes software for transmitting and receiving data through the network interface.

Brief Summary Text (25):
Accordingly, the present invention allows a designer to provide a single initialization module to users of his product in a wide variety of variant architectures. The user of the product then is relieved of the duty of selecting the particular version of the device driver. The code is automatically optimized for the architecture without user involvement and without wasting host memory space.

Drawing Description Text (2):
FIG. 1 is a flow chart of a program for initializing a computer operating system program in accordance with the present invention.

Drawing Description Text (7):
FIG. 6A is a block diagram of an initialized module for OS/2 operating system environments.

Drawing Description Text (8):
FIG. 6B is a block diagram of the initialization module for DOS operating system environments.

Drawing Description Text (9):
FIG. 7 is a flow chart of the initialization process according to a preferred embodiment of the present invention.

Drawing Description Text (10):

FIG. 8 is a flow chart of one portion of the initialization code for parsing a configuration memory image in a preferred embodiment of the present invention.

Detailed Description Text (10):
FIG. 5 is a schematic diagram of a host computer system in which the preferred embodiment of the present invention is implemented. As can be seen, the host computer system includes a host CPU 50 which is coupled to a host bus 51. Also, a host memory 52 is included connected to the host bus. The architecture further includes a mass storage device 53, such as a disk drive connected to the host bus 51. Add-on devices, such as display control 54, network interface 55, and other add-on device 56 are also coupled to the host bus 51. The network interface 55 in the preferred system is coupled to an Ethernet network. The preferred embodiment of the present invention provides a method for optimizing a device driver for the network interface 55 for an Ethernet network, according to the NDIS standard.

Detailed Description Text (11):
Two primary goals of the present invention are to create the fastest possible code for a host variant architecture and to occupy the smallest amount of host memory. Both of these goals are met through the use of the relocatable code blocks technique outlined above. In the preferred system, the technique is used for generating a driver for the NDIS industry standard network interface. Such a driver requires certain functional blocks in all of the host variant architectures. For example, one functional block is the TransmitChain procedure. In the initialization module, several different versions of the TransmitChain procedure are stored in a TransmitChain segment. One version is written for a 286 environment, one for a 386 environment, and so on. During initialization, the device driver determines its environment and selects the appropriate TransmitChain procedure for that environment. The selected procedure is then moved to the beginning of the TransmitChain segment. The rest of the TransmitChain procedures which were not appropriate for this environment are overwritten or discarded when the device driver returns from initialization.

Detailed Description Text (12):
This relocatable code block technique is implemented in the preferred system in all of the performance critical device driver code. That is, all code except for code in the SupportCode segment is implemented in this technique. Thus, many different versions of the performance critical routines are written so as to provide optimal performance and minimal size for almost any host variant architecture.

Detailed Description Text (13):
The resulting device driver, after initialization, will include the following blocks of code:

Detailed Description Text (25):
As mentioned above, when the device driver is first loaded in the memory, each of the performance critical program segments is composed of at least one code block, each code block is optimized for one or more particular variant architectures, within which the device driver is intended to run.

Detailed Description Text (26):
The initialization process for this preferred embodiment is illustrated in FIG. 7. It begins when the operating system reads the program image of the device driver from a secondary storage device and loads it into memory. The operating system then branches to the initialization code of the program, InitCodeSeg, which has been loaded into memory.

Detailed Description Text (27):
The initialization process includes the nine steps illustrated in FIG. 7.

Detailed Description Text (28):

h e b b g e e e f c e egc c e ge

As mentioned above, the process begins by loading the program and calling the initialization code (block 70). The first step involves printing a message on the display terminal identifying the software which is being initialized (block 71). Next, the device driver searches for an unused device driver name among all other device drivers which have been loaded (block 72). In the third step, the protocol manager is opened to obtain a pointer to a configuration memory image, if such configuration memory exists (block 73). Next, the configuration memory image is parsed to identify key words and parameters that inform the device driver of the host architecture (block 74).

Detailed Description Text (29):
In step 5, the expansion slots of the host computer are scanned for an adapter which will be controlled by this device driver. Once found, the adapter's configuration information is read (block 75). In step 6, the variables which could not be initialized at compile time are now initialized and code is executed which determines the device driver's host environment (block 76).

Detailed Description Text (30):
In step 7, the code blocks are selected and relocated, possibly by overwriting code blocks which are not needed. The data structures which were initialized at compile time and in step 6 (block 76) are used to guide the relocation process. The host environment which was determined in steps 4 (block 74) and 6 (block 76) is also used to guide the selection and relocation process (block 77). In step 8, the device drivers timer interrupt service routine is registered so that it will be called during each timer tick (block 78). In step 9, the device driver informs the protocol manager that it has been initialized and is able to bind with other modules in the system (block 79). Finally, the control returns to the host operating system (block 80).

Detailed Description Text (31):
As mentioned above, part of the initialization process is to scan the configuration memory image for key words and parameters (block 74). The configuration memory image, or CMI, is a structure in memory created by the protocol manager in DOS or OS/2 type systems. The protocol manager loads before the device driver of the present invention and creates the configuration memory image. The key words and parameters found in the CMI help the device driver determine its environment.

Detailed Description Text (32):
The process for parsing the configuration image is illustrated in FIG. 8. The algorithm begins at parsing the configuration memory image (block 81). In step 1, the configuration memory image structure is scanned, starting with the first module. If a module is found which contains the key word DRIVERNAME, and a parameter equal to our device driver name, then we begin examining the key words of that module, starting with the first key word (block 82). In the second step, the key word is compared against a list of valid key words. If a match is found, then the algorithm branches to step 3, otherwise, the algorithm branches to step 4 (block 83). In step 3, a procedure is called to validate the parameters of the key word and set the environment variable according to the parameters of the key word (block 84). In step 4, the program advances to the next key word in the CMI (block 85).

Detailed Description Text (33):
In step 5, if there are no more key words, then the routine returns to the main initialization routine, otherwise the process loops to step 2 (block 86). Finally, the routine returns to the main initialization module (block 87).

Detailed Description Text (34):
As mentioned above, a given host architecture may or may not include a configuration memory, or the configuration memory may not include enough information upon which the host architecture can be unambiguously identified. Thus,

the part of the initialization process, as illustrated in FIG. 7, involves executing tests to determine the host environment. In the first system, two aspects of the environment are automatically determined. The CPU type of the host machine, and whether or not the device driver can perform virtual to physical address translations under the OS/2 operating system.

Detailed Description Text (40):
Once the variant architecture of the host data processing system is determined, the relocation process may begin. This process is illustrated in FIG. 11. As shown, the code relocation process begins at block 110. The first step involves setting the destination address for relocated code blocks and examining the first functional segment (block 111). Next, it is determined which procedure code block in the segment will be relocated (block 112). The procedure code block is then moved to the destination address and the destination address is set to the end of the relocated procedure code block (block 113). Next, reference is made to the next segment if there are more segments to be reviewed (block 114). If there are more code blocks, then the algorithm loops to block 112. Otherwise, the process sets up codes to free up memory beyond the current destination address (block 115), which has been set in block 113 at the end of the last relocated procedure code block. This represents the end of the relocation process (block 116).

Detailed Description Text (41):
In the preferred system, the relocation technique is implemented using a machine environment variable and several tables. The machine environment variable is illustrated in FIG. 12. It includes a bit mask that defines the host variant architecture. All of the information that is needed to select which code block will be relocated is contained in this variable. It is stored in the ResDataSegment. The variable is compiled with a value of all zeros. The fields in it are set at various points during initialization before code relocation is done and are never changed. The fields in this bit mask variable include:

Detailed Description Text (54):
The AsynchTransferData and EarlyReceiveLookAhead functions are optimizations of the NDIS driver, the implementations of which are not important to the present invention. However, their presence in the machine environment variable indicates that this variable can be used for a variety of code optimizations for a given architecture, other than those which are host CPU and operating system dependent.

Detailed Description Text (55):
The relocation code is completely table driven. There are four types of tables illustrated in FIG. 13 which guide the relocation code through its work. These include the master relocation table 140, the index tables 141, the bounds tables 142, and the address tables 143. All of these tables are located in the InitDataSeg segment of the initialization module.

Detailed Description Text (72):
Description--This routine is called from the DOSdevInitInterrupt routine. The MachineEnvironment variable and all of the relocation tables guide this code through the relocation process. An outer loop iterates through all of the entries in the master relocation table. Each entry in the index table for the current master relocation table entry is examined in an inner loop. Once an entry is found that matches the configuration of the MachineEnvironment variable the inner loop is terminated and the corresponding entry in the bounds table is processed.

Detailed Description Text (75):
Description--This routine is called from the OS2devInitStrategy routine. The MachineEnvironment variable and all of the relocation tables guide this code through the relocation process. An outer loop iterates through all of the entries in the master relocation table. Each entry in the index table for the current master relocation table entry is examined in an inner loop. Once an entry is found

that matches the configuration of the MachineEnvironment variable the inner loop is terminated and the corresponding entry in the bounds table is processed.

Detailed Description Text (165):
In sum, a self-contained, executable module of an operating system normally stored in mass memory for use in any one of a plurality of variant host architectures is provided which is first loaded into user memory space of the central processing unit, and then executed to optimize the module for any one of the plurality of host variant architectures. It involves sensing the type of central processing unit in use, relocating a portion of the loaded executable file into an optimized segment of user accessible memory space, which may include overwriting other portions of the memory space occupied by the executable module itself, redirecting all the vectors and interrupts in accordance with the type of CPU in use and the relocated code block, and discarding all surplus segments of the executable code in the loaded module to free the memory for use by other applications programs, and/or operating system modules.

Detailed Description Paragraph Table (1):
_____ Segment: InitCodeSeg Inputs: DS points to the data segment. MachineEnvironment. All of the bits which affect code relocation must be initialized. Relocation Tables. The relocation tables must contain the proper values to guide this procedure. Outputs: Carry flag. If carry is set, then an error occurred. Otherwise, no error. DX. If an error occurred, DX contains the offset to the error message. The code segment is modified. RelocateCodeEnd. This variable is set to the offset from the beginning of the device driver to the byte beyond the last byte of code that was relocated. Interrupt Unchanged. State: Pseudo Code (copyright 1991 3COM Corporation): /* Set ES:DI to the starting location where the code will be relocated to. This is always the start of the TransmitChainSeg segment. /* mov ax,cs ; Get the current code segment mov es,ax ; Store it in ES mov di,offset Code:TransmitChainSeg ; Set DI to the offset /* In case code relocation has been disabled, set the end-of-code marker to the end of the device driver code. */ mov RelocateCodeEnd,offset Code:InitCodeSeg /* Initialize the SI register to point to the master relocation table. The number of entries in the relocation table is stored in the loop counter and SI is updated to point to the first entry in the master relocation table. */ SI = offset Code:MasterRelocationTable LoopCounter1 = word ptr [SI] If (LoopCounter1 = 0) Goto DOSRelocateCodeEnd SI = SI + 2 /* Each iteration of this loop processes one entry in the master relocation table. */ ProcessMasterRelocationTable: /* Set BX to point to the Index Table. A loop counter is initialized to contain the number of entries in the index table. */ BX = [SI].MRTIndexTablePtr LoopCounter2 = word ptr [BX] BX = BX + 2 /* Each iteration of this loop processes one entry in the index table. The loop terminates when all of the entries have been processed or an entry is found that specifies the correct bits set and clear in the machine environment variable. */ SearchIndexTable: If (((MachineEnvironment & [BX].IndexSetBits) ! = [BX].IndexSetBits) .vertline..vertline. ((MachineEnvironment & [BX].IndexClearBits) ! = 0)) /* Process the next entry. */ BX = BX + size IndexTableStruc Loop SearchIndexTable /* If we get here, we have encountered a serious error. This means that there is no code written to handle the current machine environment. THIS SHOULD NEVER HAPPEN! */ DX = &NoIndexTableEntry Set the Carry Flag return far } /* Get the bounds table index from the index table. Then set BX to the start of the bounds table. DI is aligned on a double word boundary. */ BoundsIndex = [EX].IndexBoundsIndex * size BoundsTableStruc BX = [SI].MRTBoundsTablePtr DI = (DI + size dword - 1) & not (size dword - 1) /* The current BX is saved, and BX is set to the start of the address table. A loop counter is initialized to the number of elements in the address table. */ SavedBX = BX BX = [BX+BoundsIndex].BoundsAddressTablePtr LoopCounter3 = word ptr [BX] BX = BX + 2 If (LoopCounter3 = 0) Goto RelocateCodeBlock /* Each iteration of this loop processes an entry in the address table. The appropriate location in the data segment is set to the new (relocated) location of the code block. */ ProcessAddressTable: /* Set the new offset */ DS:[[BX].AddressDGroupOffset].off =

DI + [BX].AddressBlockOffset /* If data item is a far pointer, set the code segment value. */ If ([BX.]AddressPointerType == TRUE) DS:[[BX].AddressDGroupOffset].segm = ES /* Point BX at the next address table entry and go process it. */ BX = BX + size AddressTableStruc Loop ProcessAddressTable /* It is now time to actually move the code block. The starting address and length is retrieved from the bounds table. DS and SI are temporarily used to point at the current location of the code block. */ RelocateCodeBlock: BX = SavedBX SavedSI = SI SavedDS = DS SI = [BX].BoundsStartingOffset CX = [BX].BoundsEndingOffset - SI DS = CS REP MOVSB /* Restore the DS and SI register values. Loop back to the top to process another master relocation table entry. */ DS = SavedDS SI = SavedSI + size MasterRelocationStruc Loop ProcessMasterRelocationTable /* Save the current DI value. DI points to the end of the resident code segment. */ RelocateCodeEnd = DI DOSRelocateCodeExit: Clear Carry Flag Return

Detailed Description Paragraph Table (2):

Segment: InitCodeSeg Inputs: DS points to the data segment. MachineEnvironment. All of the bits which affect code relocation must be initialized. Relocation Tables. The relocation tables must contain the proper values to guide this procedure. Outputs: Carry flag. If carry is set, then an error occurred. Otherwise, no error. DX. If an error occurred, DX contains the offset to the error message. The code segment is modified. RelocateCodeEnd. This variable is set to the offset from the beginning of the device driver to the byte beyond the last byte of code that was relocated. Interrupt Unchanged. State: Pseudo Code (copyright 1991 3COM Corporation): /* Set ES:DI to the starting location where the code will be relocated to. This is always the start of the TransmitChainSeg segment. Under OS/2 this is accomplished by converting the virtual address of the TransmitChain segment to a physical address and then converting it back to a virtual address. */ push ds ; Save DS DS = CS ; Set DS:SI to destination SI = offset Code:TransmitChainSeg DL = DEVHLP.sub.-- VIRTTOPHYS pop es ; Get DS value back call es:devHelper DS = ES CX = 65535 ; Maximum size segment DH = 1 ; Store result in ES:DI DL = DEVHLP.sub.-- PHYSTOVIRT call devHelper /* In case code relocation has been disabled, set the end-of-code marker to the end of the device driver code. */ mov RelocateCodeEnd,offset Code:InitCodeSeg /* Initialize the SI register to point to the master relocation table. The number of entries in the relocation table is stored in the loop counter and SI is updated to point to the first entry in the master relocation table. */ SI = offset DGROUP:MasterRelocationTable LoopCounter1 = word ptr [SI] If (LoopCounter1 = 0) Goto DOSRelocateCodeEnd SI = SI + 2 /* Each iteration of this loop processes one entry in the master relocation table */ ProcessMasterRelocationTable: /* Set BX to point to the Index Table. A loop counter is initialized to contain the number of entries in the index table. */ BX = [SI).MRTIndexTablePtr LoopCounter2 = word ptr [BX] BX = BX + 2 /* Each iteration of this loop processes one entry in the index table. The loop terminates when all of the entries have been processed or an entry is found that specifies the correct bits set and clear in the machine environment variable. */ SearchIndexTable: If (((MachineEnvironment & [BX].IndexSetBits) ! = [BX].IndexSetBits) .vertline..vertline. ((MachineEnvironment & [BX].IndexClearBits) ! = 0)) /* Process the next entry. */ BX = BX + size IndexTableStruc Loop SearchIndexTable /* If we get here, we have encountered a serious error. This means that there is no code written to handle the current machine environment. THIS SHOULD NEVER HAPPEN! */ DX = &NoIndexTableEntry Set the Carry Flag return } /* Get the bounds table index from the index table. Then set BX to the start of the bounds table. DI is aligned on a double word boundary. */ BoundsIndex = [BX].IndexBoundsIndex * size BoundsTableStruc BX = [SI].MRTBoundsTablePtr DI = (DI + size dword - 1) & not (size dword - 1) /* The current BX is saved, and BX is set to the start of the address table. A loop counter is initialized to the number of elements in the address table. */ SavedBX = BX BX = [BX+BoundsIndex].BoundsAddressTablePtr LoopCounter3 = word ptr [BX] BX = BX + 2 If (LoopCounter3 = 0) Goto RelocateCodeBlock /* Each iteration of this loop processes an entry in the address table. The appropriate location in the data

h e b b g e e e f c e egc c e ge

segment is set to the new (relocated) location of the code block. */ ProcessAddressTable: /* Set the new offset */ DS: [[BX].AddressDGroupOffset].off = DI + [BX].AddressBlockOffset /* If data item is a far pointer, set the code segment value. */ If ([BX].AddressPointerType == TRUE) DS:[[BX]AddressDGroupOffset].segm = ES /* Point BX at the next address table entry and go process it. */ BX = BX + size AddressTableStruc Loop ProcessAddressTable /* It is now time to actually move the code block. The starting address and length is retrieved from the bounds table. DS and SI are temporarily used to point at the current location of the code block. */ RelocateCodeBlock: BX = SavedBX SavedSI = SI SavedDS = DS SI = [BX].BoundsStartingOffset CX = [BX].BoundsEndingOffset - SI DS = CS REP MOVSB /* Restore the DS and SI register values. Loop back to the top to process another master relocation table entry. */ DS = SavedDS SI = SavedSI + size MasterRelocationStruc Loop ProcessMasterRelocationTable /* Save the current DI value. DI points to the end of the resident code segment. */ RelocateCodeEnd = DI OS2RelocateCodeExit: Clear Carry Flag Return

Other Reference Publication (1):
"Automatic Configuration of a Personal Computer System", IBM Technical Disclosure Bulletin, vol. 32, No. 4B, Sep. 1989, pp. 112-115.

CLAIMS:

1. A method for initializing operating system software for controlling an add-on device in a host data processing system, the host having any one of a plurality of variant architectures, comprising:

loading into host memory an initialize module including a set of operating system code blocks for the add-on device, each code block adapted for at least one of the plurality of variant architecture, the initialize module including tables for indentifying locations of code blocks, and of location-dependent entries within code blocks;

after loading the initialize module, executing an identifying portion of the initialize module to identify the variant architecture of the host;

after executing the identifying portion, executing a selecting portion of the initialize module to select a subset of the set of operating system code blocks adapted for the identified variant architecture, including accessing the tables in response to the identified variant architecture to identify selected code blocks;

placing the subset within the host memory in contiguous memory locations, including accessing selected code blocks and updating location-dependent entries in selected code blocks; and

freeing memory locations in host memory of the initialize module outside the contiguous memory locations.

2. The method of claim 1, wherein the set of operating system code blocks in the initialize module comprises a plurality of functional segments, each functional segment in the plurality including at least one code block, and the step of executing a selecting portion includes:

selecting a code block form each functional segment in the plurality of functional segments.

3. The method of claim 1, wherein the step of loading includes storing the initialize module in a memory space beginning at a first address and ending at a second address; and

the contiguous memory locations begin at the first address and end at a third address; and

the step of freeing includes freeing memory locations between the third and second addresses.

4. The method of claim 1, wherein at least one of the plurality of variant architectures includes a configuration table storing variables identifying attributes of the host architecture, and the initialize module includes an architecture variable, and the step of executing an identifying portion includes:

parsing the configuration table; and

updating the architecture variable in response to entries in the configuration table.

5. The method of claim 1, wherein the initialize module includes an architecture variable, and the step of executing an identifying portion includes:

executing tests to determine features of the host architecture; and

updating the architecture variable in response to results of the tests.

7. The method of claim 1, wherein host architecture includes a limited address space and the subset of code blocks for the add-on device occupies host memory during execution of applications software.

8. A method for initializing operating system software for controlling an add-on device in a host data processing system, the host having any one of a plurality of variant architectures and limited host memory, comprising:

loading into a memory space beginning at a first address and ending at a Second address in host memory an initialize module, the initialize module including an environment variable, and a plurality of functional segments, each functional segment in the plurality including at least one code block for the add-on device, each code block adapted for at least one of the plurality of variant architectures, and tables for identifying a code block in each functional segment in response to the environment variable;

executing an identifying portion of the initialize module to identify the variant architecture of the host and to update the environment variable to indicate the identified variant architecture;

executing an accessing portion of the initialize module to access the tables in response to the updated environment variable to select a code block adapted for the indicated variant architecture from each of the plurality of functional segments;

placing the selected code blocks within the host memory in contiguous memory locations beginning at the first address and ending at a third address; and

freeing memory locations of the initialize module in host memory outside the contiguous memory locations between the third address and the second address.

9. The method of claim 8, wherein the tables in the initialize module identify location-dependent entries within the selected code blocks; and the step of placing includes:

accessing selected code blocks and updating location-dependent entries in selected code blocks.

10. The method of claim 8, wherein at least one of the plurality of variant architectures includes a configuration table storing variables identifying attributes of the host architecture and the step of executing an identifying portion includes:

parsing the configuration table; and

updating the architecture variable in response to entries in the configuration table.

11. The method of claim 8, wherein the step of executing an identifying portion includes:

executing tests to determine features of the host architecture; and

updating the architecture variable in response to the results of the tests.

L31: Entry 1 of 1 File: USPT Jul 7, 1992

DOCUMENT-IDENTIFIER: US 5128995 A
TITLE: Apparatus and method for loading a system reference diskette image from a system partition in a personal computer system

Abstract Text (1):
A personal computer system according to the present invention comprises a system processor, a random access memory, a read only memory, and at least one direct access storage device. A direct access storage device controller coupled between the system processor and direct access storage device includes a protection mechanism for protecting a region of the storage device. The protected region of the storage device includes a master boot record, a BIOS image and a system reference diskette image. The BIOS image includes a section known as Power on Self Test (POST). POST is used to test and initialize a system. Upon detecting any configuration error, system utilities from the system reference diskette image, such as set configuration programs, diagnostic programs and utility programs can be automatically activated from the direct access storage device.

Brief Summary Text (11):
These systems can be classified into two general families. The first family, usually referred to as Family I Models, use a bus architecture exemplified by the IBM PERSONAL COMPUTER AT and other "IBM compatible" machines. The second family, referred to as Family II Models, use IBM's Micro Channel bus architecture exemplified by IBM's PERSONAL SYSTEM/2 Models 50 through 80.

Brief Summary Text (12):
Beginning with the earliest personal computer system of the family I models, such as the IBM Personal Computer, it was recognized that software compatibility would be of utmost importance. In order to achieve this goal, an insulation layer of system resident code, also known as "firmware", was established between the hardware and software. This firmware provided an operational interface between a user's application program/operating system and the device to relieve the user of the concern about the characteristics of hardware devices. Eventually, the code developed into a BASIC input/output system (BIOS), for allowing new devices to be added to the system, while insulating the application program from the peculiarities of the hardware. The importance of BIOS was immediately evident because it freed a device driver from depending on specific device hardware characteristics while providing the device driver with an intermediate interface to the device. Since BIOS was an integral part of the system and controlled the movement of data in and out of the system processor, it was resident on the system planar and was shipped to the user in a read only memory (ROM). For example, BIOS in the original IBM Personal Computer occupied 8K of ROM resident on the planar board.

Brief Summary Text (13):
As new models of the personal computer family were introduced, BIOS had to be updated and expanded to include new hardware and I/O devices. As could be expected, BIOS started to increase in memory size. For example, with the introduction of the IBM PERSONAL COMPUTER AT, BIOS grew to require 32K bytes of ROM.

Brief Summary Text (14):
Today, with the development of new technology, personal computer systems of the Family II models are growing even more sophisticated and are being made available to consumers more frequently. Since the technology is rapidly changing and new I/O devices are being added to the personal computer systems, modification to the BIOS has become a significant problem in the development cycle of the personal computer system.

Brief Summary Text (15):
For instance, with the introduction of the IBM Personal System/2 with Micro Channel architecture, a significantly new BIOS, known as advanced BIOS, or ABIOS, was developed. However, to maintain software compatibility, BIOS from the Family I models had to be included in the Family II models. The Family I BIOS became known as Compatibility BIOS or CBIOS. However, as previously explained with respect to the IBM PERSONAL COMPUTER AT, only 32K bytes of ROM were resident on the planar board. Fortunately, the system could be expanded to 96K bytes of ROM. Unfortunately, because of system constraints, this turned out to be the maximum capacity available for BIOS. Luckily, even with the addition of ABIOS, ABIOS and CBIOS could still squeeze into 96K of ROM. However, only a small percentage of the 96K ROM area remained available for expansion. With the addition of future I/O devices, CBIOS and ABIOS will eventually run out of ROM space. Thus, new I/O technology will not be able to be easily integrated within CBIOS and ABIOS.

Brief Summary Text (18):
It is appropriate at this time to briefly explain the purpose of the system utilities previously stored on the reference diskette. With the introduction of IBM's PS/2 Micro Channel Systems came the removal of switches and jumpers from I/O adapter cards and planar. Micro Channel Architecture provided for programmable registers to replace them. Utilities to configure these programmable registers or programmable option select (POS) registers were required. In addition, other utilities to improve system usability characteristics along with system diagnostics were shipped with each system on this system reference diskette.

Brief Summary Text (19):
Prior to initial use, each Micro Channel System required its POS registers to be initialized. For example, if the system is booted with a new I/O card, or a slot change for an I/O card, a configuration error is generated and the system boot up procedure halts. The user is then prompted to load the system reference diskette and press the F1 key. A "Set Configuration Utility" can then be booted from the system reference diskette to configure the system. The Set Configuration Utility will prompt the user for the desired action. If the appropriate I/O card's descriptor files are loaded on the system reference diskette, the Set Configuration Utility will generate the correct POS or configuration data in non-volatile storage. The descriptor file contains configuration information to interface the card to the system.

Brief Summary Text (23):
Another objective of the present invention is to improve the usability of these systems by providing each system its own personalized copy of the system reference diskette and configuration files.

Brief Summary Text (24):
Broadly considered, a personal computer system according to the present invention comprises a system processor, a random access memory, a read only memory, and at least one direct access storage device. A direct access storage device controller coupled between the system processor and direct access storage device includes a means for protecting a region of the storage device. The protected region of the storage device includes a master boot record, a BIOS image and the system reference diskette image. The BIOS image includes a section known as Power on Self Test

(POST). POST is used to test and initialize a system. Upon detecting any configuration error, system utilities from the system reference diskette image, such as set configuration programs, diagnostic programs and utility programs can be automatically activated.

Brief Summary Text (26):
In particular, the read only memory includes a first portion of BIOS. The first portion of BIOS initializes the system processor, the direct access storage device and resets the protection means to read the master boot record from the protected region or partition on the direct access storage device into the random access memory. The master boot record includes a data segment and an executable code segment. The data segment includes data representing system hardware and a system configuration which is supported by the master boot record. The first BIOS portion confirms the master boot record is compatible with the system hardware by verifying the data from the data segment of the master boot record agrees with data included within the first BIOS portion representing the system processor, system planar, and planar I/O configuration.

Brief Summary Text (27):
If the master boot record is compatible with the system hardware, the first BIOS portion vectors the system processor to execute the executable code segment of the master boot record. The executable code segment confirms that the system configuration has not changed and loads in the remaining BIOS portion from the direct access storage device into random access memory. The executable code segment then verifies the authenticity of the remaining BIOS portion, vectors the system processor to begin executing the BIOS now in random access memory. BIOS, executing in random access memory, generates the second signal for protecting the disk partition having the remaining BIOS and then boots up the operating system to begin operation of the personal computer system. The partition holding the remaining BIOS is protected to prevent access to the BIOS code on disk in order to protect the integrity of the BIOS code.

Detailed Description Text (4):
In use, the personal computer system 10 is designed primarily to give independent computing power to a small group of users or a single user and is inexpensively priced for purchase by individuals or small businesses. In operation, the system processor operates under an operating system, such as IBM's OS/2 Operating System or DOS. This type of operating system includes a BIOS interface between the DASD 12-16 and the Operating System. A portion of BIOS divided into modules by function is stored in ROM on the planar 24 and hereinafter will be referred to as ROM-BIOS. BIOS provides an interface between the hardware and the operating system software to enable a programmer or user to program their machines without an indepth operating knowledge of a particular device. For example, a BIOS diskette module permits a programmer to program the diskette drive without an indepth knowledge of the diskette drive hardware. Thus, a number of diskette drives designed and manufactured by different companies can be used in the system. This not only lowers the cost of the system 10, but permits a user to choose from a number of diskette drives.

Detailed Description Text (5):
Prior to relating the above structure to the present invention, a summary of the operation in general of the personal computer system 10 may merit review. Referring to FIG. 2, there is shown a block diagram of the personal computer system 10. FIG. 2 illustrates components of the planar 24 and the connection of the planar 24 to the I/O slots 18 and other hardware of the personal computer system. Located on the planar 24 is the system processor 26 comprised of a microprocessor which is connected by a local bus 28 to a memory controller 30 which is further connected to a random access memory (RAM) 32. While any appropriate microprocessor can be used, one suitable microprocessor is the 80386 which is sold by Intel.

Detailed Description Text (6):
While the present invention is described hereinafter with particular reference to the system block diagram of FIG. 2, it is to be understood at the outset of the description which follows, it is contemplated that the apparatus and methods in accordance with the present invention may be used with other hardware configurations of the planar board. For example, the system processor could be an Intel 80286 or 80486 microprocessor.

Detailed Description Text (8):
The local bus 28 is further connected through a bus controller 34 to a read only memory (ROM) 36 on the planar 24. An additional nonvolatile memory (NVRAM) 58 is connected to the microprocessor 26 through a serial/parallel port interface 40 which is further connected to bus controller 34. The nonvolatile memory can be CMOS with battery backup to retain information whenever power is removed from the system. Since the ROM is normally resident on the planar, model and submodel values stored in ROM are used to identify the system processor and the system planar I/O configuration respectively. Thus these values will physically identify the processor and planar I/O configuration.

Detailed Description Text (9):
The NVRAM is used to store system configuration data. That is, the NVRAM will contain values which describe the present configuration of the system. For example, NVRAM contains information describing the capacity of a fixed disk or diskette, the type of display, the amount of memory, time, date, etc. Additionally, the model and submodel values stored in ROM are copied to NVRAM whenever a special configuration program, such as SET Configuration, is executed. The purpose of the SET Configuration program is to store values characterizing the configuration of the system in NVRAM. Thus for a system that is configured properly, the model and submodel values in NVRAM will be equal respectively to the model and submodel values stored in ROM. If these values are not equal, this indicates that the configuration of the system has been modified. Reference is made to FIG. 6D, where this feature in combination with loading BIOS is explained in greater detail.

Detailed Description Text (12):
Previous to the present invention, ROM 36 could include all of the BIOS code which interfaced the operating system to the hardware peripherals. According to one aspect of the present invention, however, ROM 36 is adapted to store only a portion of BIOS. This portion, when executed by the system processor 26, inputs from either the fixed disk 62 or diskette 66 a second or remaining portion of BIOS, hereinafter also referred to as a BIOS image. This BIOS image supersedes the first BIOS portion and being an integral part of the system is resident in main memory such as RAM 32. The first portion of BIOS (ROM-BIOS) as stored in ROM 36 will be explained generally with respect to FIGS. 3-4 and in detail with respect to FIGS. 6A-D. The second portion of BIOS (BIOS image) will be explained with respect to FIG. 5, and the loading of the BIOS image with respect to FIG. 7. Another benefit from loading a BIOS image from a DASD is the ability to load BIOS directly into the system processor's RAM 32. Since accessing RAM is much faster than accessing ROM, a significant improvement in the processing speed of the computer system is achieved. An additional advantage is also gained by storing system utilities on the DASD. When a condition for the usage of the system utilities is required, the system utility can automatically be referenced on the DASD.

Detailed Description Text (13):
The explanation will now proceed to the operation of the BIOS in ROM 36 and to the operation of loading the BIOS image and system reference diskette image from either the fixed disk or diskette. In general, a first program such as ROM-BIOS prechecks the system and loads a BIOS master boot record into RAM. The master boot record includes a data segment having validation information and, being a loading means, a code segment having executable code. The executable code uses the data information to validate hardware compatibility and system configuration. After testing for

hardware compatibility and proper system configuration, the executable code loads the BIOS image into RAM producing a main memory resident program. The BIOS image succeeds ROM-BIOS and loads the operating system to begin operation of the machine. For purposes of clarity, the executable code segment of the master boot record will be referred to as MBR code while the data segment will be referred to as MBR data.

Detailed Description Text (14):
Referring to FIG. 3 there is a memory map showing the different code modules which comprise ROM-BIOS. ROM-BIOS includes a power on self test (POST) stage I module 70, an Initial BIOS Load (IBL) Routine module 72, a Diskette module 74, a hardfile module 76, a video module 78, a diagnostic-panel module 80, and hardware compatibility data 82. Briefly, POST Stage I 70 performs system pre-initialization and tests. The IBL routine 72 determines whether the BIOS image is to be loaded from disk or diskette, checks compatibility and loads the master boot record. Diskette module 74 provides input/output functions for a diskette drive. Hardfile module 76 controls I/O to a fixed disk or the like. Video module 78 controls output functions to a video I/O controller which is further connected to a video display. Diagnostic panel module 80 provides control to a diagnostic display device for the system. The hardware compatibility data 82 includes such values as a system model and submodel values which are described later with respect to FIG. 5.

Detailed Description Text (15):
Referring now to FIG. 4, there is shown a process overview for loading a BIOS image into the system from either the fixed disk or the diskette. When the system is powered up, the system processor is vectored to the entry point of POST Stage I, step 100. POST Stage I initializes the system and tests only those system functions needed to load BIOS image from the selected DASD, step 102. In particular, POST Stage I initializes the processor/planar functions, diagnostic panel, memory subsystem, interrupt controllers, timers, DMA subsystem, fixed disk BIOS routine (Hardfile module 76), and diskette BIOS routine (Diskette module 74), if necessary.

Detailed Description Text (16):
After POST Stage I pre-initializes the system, POST Stage I vectors the system processor to the Initial BIOS Load (IBL) routine included in the Initial BIOS Load module 72. The IBL routine first, determines whether the BIOS image is stored on fixed disk or can be loaded from diskette; and second, loads the master boot record from the selected media (either disk or diskette) into RAM, step 104. The master boot record includes the MBR data and the MBR code. The MBR data is used for verification purposes and the MBR code is executed to load in the BIOS image. A detailed description of the operation of the IBL routine is presented with respect to FIGS. 6A-D.

Detailed Description Text (17):
With continuing reference to FIG. 4, after the IBL routine loads the master boot record into RAM, the system processor is vectored to the starting address of the MBR code to begin execution, step 106. The MBR code performs a series of validity tests to determine the authenticity of the BIOS image and to verify the configuration of the system. For a better understanding of the operation of the MBR code, attention is directed to FIG. 7 of the drawings wherein the MBR code is described in greater detail. On the basis of these validity tests, the MBR code loads the BIOS image into RAM and transfers control to the newly loaded BIOS image in main memory, step 108. In particular, the BIOS image is loaded into the address space previously occupied by ROM-BIOS. That is if ROM-BIOS is addressed from E0000H through FFFFFH, then the BIOS image is loaded into this RAM address space thus superseding ROM-BIOS. Control is then transferred to POST Stage II which is included in the newly loaded BIOS image thus abandoning ROM-BIOS. POST Stage II, now in RAM, initializes and tests the remaining system in order to load the operating system boot, steps 110-114. Before Stage II POST transfers control to the operating system, Stage II POST sets a protection means for preventing access to

the disk partition holding the BIOS image. However, if an error is detected, Stage II POST can disable the protection means and invoke the system utilities in the system reference diskette image on the disk. Reference is made to FIGS. 8-10 for a detailed discussion of this protection process. It is noted that during a warm start, the processor is vectored to step 108, bypassing steps 100-106.

Detailed Description Text (18):
For clarity, it is appropriate at this point to illustrate a representation for the format of the master boot record. Referring to FIG. 5, there is shown the master boot record. The boot record includes the executable code segment 120 and data segments 122-138. The MBR code 120 includes DASD dependent code responsible for verifying the identity of the ROM-BIOS, checking that the IBL boot record is compatible with the system, verifying the system configuration, and loading the BIOS image from the selected DASD (disk or diskette). The data segments 122-138 include information used to define the media, identify and verify the master boot record, locate the BIOS image, and load the BIOS image.

Detailed Description Text (19):
The master boot record is identified by a boot record signature 122. The boot record signature 122 can be a unique bit pattern, such as a character string "ABC", in the first three bytes of the record. The integrity of the master boot record is tested by a checksum value 132 which is compared to a computed checksum value when the boot record is loaded. The data segments further include at least one compatible planar ID value 134, compatible model and submodel values 136. The master boot record's planar ID value defines which planar that the master boot record is valid for. Similarly, the master boot record's model and submodel values define the processor and planar I/O configuration respectively that the master boot record is valid for. It is noted that the boot record's signature and checksum identify a valid master boot record, while the boot record's planar ID, boot record's model and boot record's submodel comparisons are used to identify a boot record compatible with the system and to determine if the system configuration is valid. Another value, boot record pattern 124 is used to determine the validity of the ROM-BIOS. The boot record pattern 124 is compared to a corresponding pattern value stored in ROM. If the values match this indicates that a valid ROM-BIOS has initiated the load of a BIOS image from the selected media.

Detailed Description Text (26):
Checksum value (132): The checksum value of the data segment is initialized to generate a valid checksum for the record length value (1.5k bytes) of the master boot record code.

Detailed Description Text (29):
MBR Map length (138): The IBL map length is initialized to the number of media image blocks. In other words, if the BIOS image is broken into four blocks, the map length will be four indicating four block pointer/length fields. Usually this length is set to one, since the media image is one contiguous 128k block. MBR Media Sector Size (138): This word value is initialized to the media sector size in bytes per sector.

Detailed Description Text (33):
Referring now to FIG. 6A, under normal circumstances the system will contain a system fixed disk which the IBL routine initializes, step 150. Assume for purposes of illustration that the fixed disk is configured for Drive C of the personal computer system. Similarly, assume Drive A is designated as the diskette drive. The IBL routine then examines Drive C to determine whether it contains IBL media, step 152. Attention is directed to FIG. 6B which describes in detail this process. The IBL routine starts reading from the fixed disk at the last three sectors and continues reading, decrementing the media pointer, for 99 sectors or until a valid master boot record is found. If a master boot record is found, it is checked for system planar and processor compatibility, step 156. If it is not planar or

processor compatible, then an error is reported, step 158. Referring back to step 152, if no master boot record is found on the last 99 sectors of the fixed disk (primary hardfile), an error is reported, step 154.

Detailed Description Text (34):
Referring back to step 156, if a master boot record is found, a series of validity checks are performed to determine if the master boot record is compatible with the computer system. Additionally, the configuration of the system is checked. Attention is directed to FIG. 6D which discloses this process in greater detail. If the boot record is compatible with the planar ID, model and submodel, and if furthermore the system configuration has not changed the master boot record is loaded and the code segment of the master boot record is executed, step 160.

Detailed Description Text (36):
Referring back to step 162, if a valid password in NVRAM is not present, thus allowing BIOS image to be loaded from diskette, the IBL routine initializes the diskette subsystem, step 166. The IBL routine then determines if Drive A includes the IBL media on a diskette, step 168. If Drive A does not include IBL media, an error is generated to notify the user that an invalid diskette has been inserted in the drive, step 170. The system then halts, step 172. Attention is directed to FIG. 6C for a more detailed discussion of step 168.

Detailed Description Text (44):
Finally, FIG. 6D shows how the IBL routines tests for system planar and processor compatibility and for a proper system configuration. The master boot record is checked for compatibility with the system planar by comparing the boot record planar ID value to the system planar ID read by the system processor, step 260. If the system planar ID does not match the boot record planar ID value, this indicates this master boot record is not compatible with this planar. An error is indicated and control returns to the IBL routine, steps 262, 264, and 266.

Detailed Description Text (45):
If the master boot record is compatible with the planar, the master boot record is checked for compatibility with the processor, step 268. The boot record model value and submodel value are compared to the model value and submodel value stored in ROM respectively. A mismatch indicates a new processor has probably been inserted and this boot record is not compatible with the new processor. An error is indicated and control returned to the IBL routine, steps 270, 264 and 266. If the master boot record is compatible with the planar and processor, the process checks to determine if NVRAM is reliable, step 272. If NVRAM is unreliable, an error is indicated and control returned to the IBL routine, steps 274 and 266. If NVRAM is reliable, the system configuration is checked, step 276. A change in system configuration is indicated if the model and submodel values stored in NVRAM do not match the model and submodel values stored in ROM. Note that this last comparison will only indicate a configuration error. If a configuration error is indicated, an error is generated for the user. This error notifies the user that the configuration of the system has changed since the last time SET Configuration was run. The user is notified of the changed configuration and control passed back to the IBL routine steps 278, 264, and 266. This error is not fatal itself, but notifies the user that SET Configuration (configuration program) must be executed. Referring back to step 276, if the system model/submodel values match, an indication of comparability is set and the routine returns, steps 276, 274 and 266. Thus, the compatibility between the master boot record and the system are tested along with determining if the system configuration has been modified.

Detailed Description Text (46):
After the IBL routine loads the master boot record into RAM, it transfers control to the MBR code starting address. Referring to FIG. 7, the executable code segment of the master boot record first verifies the boot record pattern to the ROM pattern, step 300. If the pattern in the master boot record does not match the

pattern in ROM, an error is generated and the system halts, steps 302 and 305. The check for equality between ROM and boot record patterns ensures that the master boot record loaded from either the disk or diskette is compatible with the ROM on the planar board. Referring back to step 300, if the pattern in ROM matches the pattern in the boot record, the MBR code compares the system planar ID value, model and submodel value against the corresponding master boot record values, step 304. This process was discussed in greater detail with respect to FIG. 6D. If the values don't match, the master boot record is not compatible with the system planar and processor, or the system configuration has changed, and an error is generated, step 306. The system will halt when the IBL record is incompatible with planar, model or submodel values, step 305.

Detailed Description Text (48):
Referring to FIG. 8, there is shown a block diagram of an intelligent disk controller 350 for controlling movement of data between the disk drive 351 and the system processor. It is understood that disk controller 350 can be incorporated into the adapter card 60 while disk drive 351 can be included onto drive 62 of FIG. 2. A suitable disk controller 350 is a SCSI Adapter having a part number of 33F8740, which is manufactured by International Business Machines Corporation. It is understood that the disk controller 350 includes a microprocessor 352 operating under its own internal clock, for controlling its internal operations as well as its interfacing with the other elements of the disk subsystem and the system processor. The microprocessor 352 is coupled by a instruction bus 354 to a read only memory (ROM) 356 which stores instructions which the disk controller 350 executes to process and control the movement of data between the disk drive and the system processor. It is also understood that disk controller 350 can include random access memory coupled to microprocessor 352 for the storage or retrieval of data. The movement of data between disk controller 350 and the system processor is effected by data bus 358 and instruction bus 360. A reset signal on line 362 resets or initializes the disk controller logic upon power-on sequence or during a system reset. The reset signal is generated by the planar board logic, and can take the form of a channel reset signal as provided by IBM's Micro Channel architecture as described in "IBM PERSONAL SYSTEM/2 Seminar Proceedings", Volume 5, Number 3, May 1987 as published by the International Business Machines Corporation Entry Systems Division. Furthermore, the reset signal can be effectively initiated by BIOS outputting a particular bit configuration to an I/O port of the system processor in which the planar logic is connected.

Detailed Description Text (50):
Referring now to FIG. 9, there is shown a flowchart diagramming the read, write, and protect functions of the disk controller which are effected by the operation of routines stored in ROM 356. In operation, a disk instruction is initiated by the system processor and transferred to the disk controller 350. The disk controller receives and interprets the instruction to perform the designated operation, step 400. The disk controller first determines if this is a write operation in which data from the system processor are stored on the disk drive hardware, step 402. If the instruction is a write instruction, data are received from the system processor in relative block address (RBA) format.

Detailed Description Text (57):
Referring back to step 416, if the instruction is not a write or read instruction, it is tested for a set maximum RBA instruction, step 428. This instruction allows the disk controller to create a protectable area or partition on the disk drive hardware. This instruction allows the disk controller to set M between 0 and N blocks, step 430. It is important to note that when the disk controller is reset (through the reset signal) that M is set so that the maximum number of blocks are available. That is, when the disk controller is reset, M=N. Essentially, protection for the protectable area is eliminated upon resetting the disk controller, allowing access to the area. However, once the set maximum RBA instruction is executed only a reset or another set maximum RBA instruction will allow access to the protectable

area. Conceptually, the setting of the maximum RBA can be thought of as setting a fence which protects access to the area above the fence while allowing access to the area below the fence. The disk controller then returns to wait for another instruction, step 412.

Detailed Description Text (60):
Referring now to FIG. 10, there is shown a block flow diagram effecting the protection of the IBL media. From a power-on condition the system is initialized and BIOS initiates activity in planar board logic to send a reset condition to the disk controller, steps 450 and 452. The reset signal drops the fence and allows the system processor to access the IBL media previously stored on the disk in the area from M blocks to N blocks. The system loads the IBL media as previously described with reference to FIG. 4-7, step 454. During the IBL loading sequence Post Stage II is executed, step 456. One of the tasks of POST Stage II is to execute the set maximum RBA instruction with the maximum RBA set to the first block of the IBL media which is designated as M, step 458. M is dependent upon partition type (none, partial or full) as previously explained. This in effect sets the fence denying access to the IBL media while allowing access to other regions of the disk. The operating system is then booted up in a normal fashion, step 460.

Detailed Description Text (70):
The Bootstrap Loader is used to select the appropriate boot device and read in the boot record from the active partition. The priority of the boot drives are the first diskette drive followed by the first fixed disk, such as the boot fixed disk. However, the priority of the default boot device sequence can be changed by using a utility on the system reference diskette or system reference diskette image in the system partition. The Bootstrap Loader then turns control over to the executable code in the boot record. This in turn boots the desired operating system or control program.

Detailed Description Text (83):
Thus, there has been shown a method and apparatus for booting the system reference diskette image from the system partition from a mass storage device, such as a fixed disk drive. The system partition is provided by protecting an area on the disk drive. The system partition is made bootable by storing the starting address of system partition on the disk drive and indicating to BIOS to use this as the fixed disk origin when a boot of the system reference diskette image is requested or required. By providing this capability, the system reference diskette utilities are automatically available any time the configuration is changed, a system utility is desired or an error is encountered during the execution of POST. Thus enhancing the usability of the system.

CLAIMS:

1. An apparatus for protecting system utilities in a personal computer system, the personal computer system having a system processor for executing an operating system, a read only memory, a random access memory, and at least one direct access storage device, said apparatus comprising:

a direct access storage device controller having a protection means for protecting a region of the direct access storage device, said protection means allowing access to the protected region in response to a reset signal;

a portion of BIOS being included in the protected region of the direct access storage device, said portion of BIOS being loaded into random access memory to boot the operating system, said portion of BIOS activating said protection means to prevent access to the protected region of the direct access storage device during normal operations of the operating system; and

a portion of system utilities included in the protected region of the direct access

storage device, said system utilities being automatically executed upon detecting an error condition in the loading of the operating system and further wherein said portion of system utilities comprises a program for modifying the configuration of the system.

3. An apparatus for protecting a system utility program in a personal computer system, the personal computer system having a system processor, a read only memory, a main memory, and at least one direct access storage device capable of storing a plurality of data records, said apparatus comprising:

a first program being included in the read only memory, said first program initializing the system processor, said first program further initiating the generation of a reset signal to the direct access storage device to permit access to the data records;

a loading means for loading data records from the direct access storage device into main memory, said loading means being stored in a protectable partition of the direct access storage device, said loading means being read from the direct access storage device into main memory by said first program, wherein said first program activates said loading means;

a main memory resident program image being stored in the protectable partition of the direct access storage device, said main memory resident program image being read from the direct access storage device into main memory by said loading means to produce a main memory resident program;

means for protecting the protectable partition of the direct access storage device, said protection means being activated by said main memory resident program to prevent unauthorized access to said loading means and said main memory resident program image; and a system utility program image being stored in the protectable partition of the direct access storage device, said system utility program being read automatically from the direct access storage device into main memory for execution upon said main memory resident program detecting an error in the system wherein said system utility program image includes a program for modifying the configuration of the system.